IDA

INSTITUTE FOR DEFENSE ANALYSES

Nearest Neighbor Search Applications for the Terasys Massively Parallel Workstation

Eric W. Johnson

August 1996

Approved for public release; distribution unlimited.

IDA Paper P-3169

Log: H 96-002880

PREFACE

The work reported in this document was carried out under the Independent Research Program of the Institute for Defense Analyses (IDA) as Central Research Project 9000-605, Applications of Parallel Processing. The project was conducted in IDA's Computer and Software Engineering Division.

The author would like to thank the following people for help with this project and with this report: Alfred E. Brenner, Anne A. Douville, Edward A. Feustel, Michael C. Frame, Michael T. Goodrich, David S. Hough, Norman R. Howes, Reginald N. Meeson, and Richard P. Morton. The author would like to thank Harold E. Conn for help with Terasys programming and for running the Terasys tests described in this report. The author would like to thank James R. Silk for suggesting how nearest neighbor search could be applied to radar analysis.

Table of Contents

SUMMARY

The Terasys workstation is a massively parallel computer developed at IDA's Center for Computing Sciences (formerly the Supercomputing Research Center) in Bowie, Maryland. The Terasys workstation is a SIMD (Single Instruction, Multiple Data) computer, meaning that all processors perform identical instructions and that each processor has its own local memory. Terasys processors are single-bit processors, meaning, for example, that operations like adding two multi-bit integers require several processor cycles. A Terasys workstation consists of a linear array of Terasys processors controlled by a Sun Microsystems Sparcstation-2 host. The Center for Computing Sciences has built working Terasys prototypes with up to 32,768 processors.

The existence of working Terasys prototypes at the Center for Computing Sciences made it appropriate to investigate ways in which the computing power of the Terasys could be applied to help meet the needs of IDA sponsors in divisions other than the Center for Computing Sciences. After discussing potential Terasys applications with a number of IDA researchers, we decided to focus on Terasys applications involving nearest neighbor search. Nearest neighbor search can be implemented efficiently on the Terasys and there are a number of applications of nearest neighbor search that are potentially of interest to IDA sponsors. One particularly important application of nearest neighbor search is nearest neighbor classification--classifying new samples by finding previously classified samples that they are similar to.

To implement nearest-neighbor search on the Terasys, one training record can be placed in each processor and the host can be used to broadcast each sample record. Each time a sample is broadcast, the sample can be compared with many training records in parallel. In the report we present experimental results showing that a Terasys with 32,768 processors can perform a particular nearest neighbor search problem up to 69 times faster than a 61 MHz Sun Sparcstation-20.

For certain types of records and distance functions, and particularly for nearest neighbor search in a Euclidean space, it may be more efficient to place adjacent record components in adjacent processors. In such a case, we refer to the sequence of processors used to store a record as a *vector comparison unit.*

To illustrate how Terasys-based nearest neighbor search could be applied to the work of IDA sponsors outside of the Center for Computing Sciences, the report discusses three possible applications of nearest neighbor search to an ongoing radar evaluation project in IDA's Science and Technology Division.

This report demonstrates that nearest neighbor search can be implemented efficiently on the Terasys and that nearest neighbor search can be applied to a number of problems that are potentially of interest to IDA sponsors. We recommend that the next step in this investigation should be additional experiments which use Terasys-based nearest neighbor search in a variety of applications. We particularly recommend tests that would use the Terasys to assist in the development of new nearest neighbor classifiers.

1. INTRODUCTION

The Terasys workstation is a massively parallel computer developed at IDA's Center for Computing Sciences (formerly the Supercomputing Research Center) in Bowie, Maryland. The Terasys workstation is a SIMD (Single Instruction, Multiple Data) computer, meaning that all processors perform identical instructions and that each processor has its own local memory. Terasys processors are single-bit processors, meaning, for example, that operations like adding two multi-bit integers require several processor cycles. A Terasys workstation consists of a linear array of Terasys processors controlled by a Sun Microsystems Sparcstation-2 host. The Center for Computing Sciences has built working Terasys prototypes with up to 32,768 processors.

The existence of working Terasys prototypes at the Center for Computing Sciences made it appropriate to investigate ways in which the computing power of the Terasys could be applied to help meet the needs of IDA sponsors in divisions other than the Center for Computing Sciences. After discussing potential Terasys applications with a number of IDA researchers, we decided to focus on Terasys applications involving nearest neighbor search. Nearest neighbor search can be implemented efficiently on the Terasys and there are a number of applications of nearest neighbor search that are potentially of interest to IDA sponsors. One particularly important application of nearest neighbor search is nearest neighbor classification--classifying a new sample by finding the previously classified sample that is most similar to it.

Given a set S of sample objects and a set T of training objects, the goal of nearest neighbor search is to find the nearest neighbor in T of every object in S. Nearest neighbor search problems can involve many kinds of objects and many different distance measures. For example:

1. Sets S and T might contain database records each describing a particular object or event. The goal might be to classify each object in S by finding the object in T that is most similar to it. In this case the distance between two records could be an arbitrary function involving the fields in each record.

2. A number of pattern recognition techniques make use of nearest neighbor search. For example, suppose that S and T contain vectors representing time series. The vectors in T might correspond to patterns that are of particular interest for some application and the goal might be to identify vectors in S that are similar to vectors in T. The distance between two time series might be the sum of the products of the two series at each point in time.
3. As described in [Stan 1986], each element in S and T might be a set of English words converted into a bit vector by computing multiple hash values for each word. If the vectors in T contain key words from abstracts of articles, the goal might be to find abstracts in T that share many key words with queries in S.
4. Sets S and T might contain points in a high-dimensional Euclidean space and the distance between two points could be the ordinary Euclidean distance.

This report provides an overview of the Terasys workstation and discusses how nearest neighbor search can be implemented on the Terasys. The report presents experimental results showing that a Terasys with 32,768 processors can perform a particular nearest neighbor search problem up to 69 times faster than a 61 MHz Sun Sparcstation-2.

2. NEAREST NEIGHBOR CLASSIFICATION

A typical problem which a credit card company might face is the task of predicting how satisfied it would be with a new applicant. Suppose that the company maintains a database of information about existing customers. A natural way for the company to predict how satisfied it would be with a new applicant is to find out how satisfied it has been with existing customers who are similar to the new applicant.

Predicting the behavior of new customers by finding similar existing customers is an example of *nearest neighbor classification* [Fix 1951, 1952], [Cove 1967]. Nearest neighbor classification is closely related to a technique called *memory-based reasoning* [Cree 1992].

Consider a sample record R and a set T of correctly classified training records. The m-nearest neighbor classification rule classifies R by performing the following two steps:

1. Find the m records $U_1, \ldots, U_m$ in T which are closest to R using some sort of distance measure.
2. Assign R to the class which appears the most frequently among $U_1, \ldots, U_m$.

As the following example illustrates, the definition of what it means for two records to be similar may vary from problem to problem.

In the credit card problem discussed earlier, the following information might be available for each new applicant:

1. id (integer)
2. age (integer)
3. income (integer)
4. occupation (discreet: 1 = student, 2 = self-employed, . . .)
5. credit_rating (integer)

Suppose that the above information is also available for all existing and previous customers. In addition, suppose that each existing or previous customer has been classified as either satisfactory or unsatisfactory, based on the company's experience with them.

Assuming that age, income, and credit rating have all been normalized on a scale of 0 to 10, the following algorithm computes a value between 0 and 100 indicating the distance between two records. The algorithm gives 50% of total weight to income, 30% to credit rating, and 10% each to age and job type.

```
procedure ComputeDistance(record R1, record R2)
    distance = 0;
    distance = distance + (5 * abs(R1.income - R2.income));
    distance = distance + (3 * abs(R1.credit_rating - R2.credit_rating));
    distance = distance + abs(R1.age - R2.age);
    if (R1.occupation ≠ R2. occupation)
        distance = distance + 10;
    return distance;
```

(In the above procedure, function "*abs*" returns the absolute value of an integer.)

Besides its intuitive appeal, nearest neighbor classification has several other advantages. New "knowledge" can be added to a classifier simply by adding more training records. A nearest neighbor classifier can be tuned by modifying distance functions or by adjusting the number of nearest neighbors used. Classifier results can be justified or explained by showing a user the training records used in the classification.

A primary disadvantage of nearest neighbor classification is that for large data sets, the computation time required to find nearest neighbors can be substantial. One way to speed the process up is to use a parallel computer such as the Terasys to perform many comparisons simultaneously.

3. OVERVIEW OF THE TERASYS WORKSTATION

As has already been discussed, a Terasys workstation consists of a linear array of single-bit SIMD processors connected to a Sun Microsystems Sparcstation-2 host. Current Terasys prototypes have up to 32,768 processors.

The Terasys workstation is called a workstation because it is intended to be small enough and affordable enough to potentially be dedicated to a single analyst. Gokhale et al. [Gokh 1995] estimate that in limited commercial production the chips required for a 32K processor Terasys array might cost approximately $32,000; including the Sparcstation host and other items needed to build the Terasys might bring the final price to approximately $42,000.

The Terasys is based on the PIM (Processor-in-Memory) chip. As the name suggests, a PIM chip can be thought of as a conventional memory chip in which some of the area that would have been used for storage has instead been used to implement 1-bit processors. Each PIM chip currently contains 64 processors and each processor has 2,048 bits of local memory. This means that a single PIM chip has 128K bits of storage.

The Terasys architecture is designed to be extensible and Terasys (or Petasys) computers containing substantially more than 32K processors could potentially be built.

3.1 Interprocessor Communication

In a Terasys workstation, the ability of the host to broadcast a message to the entire processor array is implicit in the dbC language (described in the next section) used to program the Terasys. If *a* is a parallel variable and *b* is a host variable, the value of *b* can be broadcast to all Terasys processors simply by executing the statement "$a = b$". In addition to implicit communication using assignment statements, the Terasys architecture also permits the following kinds of communication:

1. The host can read and write to the memory of individual processors.
2. Special instructions allow the host to efficiently obtain the maximum or minimum value of a parallel variable and the ID of the processor which holds the value. We will refer to these instructions as *global min* and *global max* operations.

3. The Terasys processor array supports a parallel prefix communication network which allows values in particular processors to be broadcast to their n left-most neighbors where n is an arbitrary power of 2. One important application of the parallel prefix network is in efficiently computing sums over groups of processors. The linear array of Terasys processors can be divided into equal-length segments where the length of each segment is an arbitrary power of 2. If x is a parallel variable, the parallel prefix network can be used to compute the sum of x over each segment in time proportional to the logarithm of the length of the segment. When the computation is complete, the value of the sum is located in the final processor in each segment.

A Terasys workstation also has the ability to transfer properly formatted data back and forth from host to Terasys memory at roughly the same speed that data can be transferred within host memory. When data is transferred from host memory to Terasys memory in this way, a 32-bit word of host data is not written to the memory of a single processor. Instead, each bit in the 32-bit word is written to a bit in one of 32 contiguous processors. In order for efficient data transfer from host to Terasys to take place in this way, the data in the host must be "corner turned" or preprocessed into a one bit per processor format.

Most Terasys applications use integer arithmetic. While Terasys processors can perform floating point arithmetic (operating on just one bit per cycle), most floating pointing operations are too slow to be of practical use on a Terasys with 32K processors.

Parallel arrays are supported on a Terasys workstation with the important restriction that Terasys processors do not support indirect addressing. This means that when array operations are performed in parallel, every processor must operate on the same array location (in its local memory) at the same time.

3.2 The dbC Programming Language

A Terasys workstation can be programmed using a language called dbC (data-parallel, bit C) developed at the Center for Computing Sciences. The dbC language is an extension of C which allows integrated programming of both the host computer and the processor array. The dbC language allows variables to be declared as type *poly*, meaning that a copy of the variable is placed on each processor. Variables which are not declared to be of type *poly* exist only on the host. Consider the following lines of dbC code:

```
poly int c;
int i;

c = 0;
for (i = 0; i < 10; i++)
    c++;
```

Assuming that it is run on a 32K processor Terasys workstation, the above program fragment would execute as follows. The declaration "*poly int c*" means that a copy of variable *c* is placed on each of the 32K processors. The declaration "*int i*" means that one copy of variable *i* is placed on the host computer. Executing the statement "*c* = 0" causes each of the 32K processors to initialize its copy of *c* to 0 in parallel. Each time the host computer executes a cycle of the "*for*" loop, all 32K copies of variable *c* are incremented in parallel.

The fact that Terasys processors are bit-serial means that they are well-suited for performing bit-oriented operations. The dbC language includes facilities for declaring and manipulating parallel arrays of bits and parallel integers with an arbitrary number of bits.

The dbC language makes it possible to write code that executes on only a subset of the Terasys processors. Consider the following statements:

```
poly int c;

c = 0;
if (DBC_iproc >= 16384)
    c++;
```

In the above lines of code, *DBC_iproc* is a predefined parallel variable which tells each processor what its ID is. Executing the "*if*" statement in the above code fragment on a 32K processor Terasys would cause the processors with IDs between 16,384 and 32,767 to increment their copies of *c* in parallel. The processors with IDs between 0 and 16,383 would remain idle.

4. USING THE TERASYS TO FIND NEAREST NEIGHBORS

Consider a set *S* of sample records and a set *T* of training records. Assume that our goal is to find the *m* nearest neighbors in *T* of each record in *S*. We will assume that each record is small enough to fit in the memory of a single processor.

In general, a Terasys implementation of nearest neighbor search can either place sample records in the Terasys and broadcast training records or place training records in the Terasys and broadcast sample records.

4.1 Placing Sample Records in the Terasys

The case where samples are stored in the Terasys array might be implemented as follows. A list of nearest neighbors for each sample is maintained by the Terasys host. Each Terasys processor stores the distance from its sample to the *m*th closest training record that has been broadcast so far. When a new training record is broadcast, each processor computes the distance from its sample to the new record and checks whether the new record is closer than the sample's current *m*th nearest neighbor. If the new record is closer, the processor alerts the host which then updates the nearest neighbor list for the sample. Once an alert has been posted, the host must also update the distance to the processor's *m*th nearest neighbor.

One drawback to the approach just described is the amount of sequential processing required during the early stages of the algorithm. Since the first training record broadcast will be a nearest neighbor for every processor, the host will initially have to respond to alerts from every processor. One way to help alleviate this problem might be to have processors alert the host only when the distance to a new training record is smaller than some threshold.

If space permits, another way to avoid this problem might be to have each Terasys processor maintain its own list of *m* nearest neighbors. Since Terasys processors do not support indirect addressing, inserting new elements into these lists would require the parallel execution of *m* instructions each of the form "if you have found a new nearest neighbor and if the current element is your *m*th nearest neighbor, then overwrite that element."

4.2 Placing Training Records in the Terasys

The alternative to placing sample records on the Terasys and broadcasting training records is to place training records on the Terasys and broadcast sample records.

If enough processors are available, one training record can be placed in each processor. In this case the nearest neighbor of a sample can be located by performing a single parallel distance computation followed by a Terasys global min operation. A Terasys workstation with one training record stored per processor could potentially be used as a real-time classifier.

The following algorithm handles the case where there are more training records than processors by dividing the training set into subsets which are small enough to fit into the processor array. In describing the algorithm, we initially assume that S and T are small enough to fit in the host's main memory.

During its initialization phase, the algorithm performs the following steps:

1. Load S and T from secondary storage to host memory.
2. Divide T into subsets $T_1, \ldots, T_n$ where each subset T_i is small enough to fit in the Terasys processor array.
3. Load the records in T_1 from host memory to the processor array, putting a single training record in each processor's memory.

After the initialization phase is complete, the algorithm loops through the following steps:

4. Let T_i be the subset of training records currently in the Terasys array. Each sample record in S is compared with all the records in T_i in parallel. After each parallel comparison has been performed, the host computer uses the Terasys global min feature to find the training records which are closest to the current sample. The host computer is responsible for keeping track of the m overall closest training records for each sample.
5. If all of the subsets $T_1, \ldots, T_n$ have been processed, the algorithm is complete. Otherwise, the next training subset is loaded into the Terasys processor array and the algorithm loops back to step 4.

4.3 Handling Sets That Are Too Large for Primary Storage

In the algorithm discussed in the previous section, we refer to the loop described by steps 4 and 5 as the *Terasys inner loop*. In a case where sets S and T are too large to be stored in host memory, an additional loop must be added to the algorithm during which records are loaded from secondary to host memory.

Consider an implementation of the algorithm with the constraint that only a total of c_1 sample records and c_2 training records can be stored in host memory at any given time. In this case, we divide S into subsets each of size c_1 and T into subsets each of size c_2. We embed the Terasys inner loop within a new loop which operates as follows:

1. Load the first subset of c_1 sample records from secondary to host memory.
2. Load the first subset of c_2 training records from secondary to host memory.
3. Execute the Terasys inner loop using the sample and training records currently in host memory.
4. Continue as follows:
    - If additional subsets of training records remain to be loaded from secondary to host memory, load the next training subset and loop back to step 3.
    - If additional subsets of sample records remain to be loaded from secondary to host memory, load the next sample subset and loop back to step 2.
    - If all sample records in secondary storage have been processed, the algorithm is complete.

4.4 Allocating Space in the Extended Algorithm

In the extended algorithm just described, only c_1 sample records and c_2 training records are stored in host memory at any given time. Each time a new subset of sample records is loaded into host memory, all of the training records must be sequentially loaded into host memory and processed. Consequently, if set S has been divided into k subsets of size c_1, then each training record must be loaded into host memory k times.

Because the extended algorithm must load training vectors into host memory multiple times, it is advantageous for an implementation of the algorithm to divide S into as few subsets as possible. In general, each training subset should be just big enough to fill the Terasys processor array while each sample subset should be as large as host memory permits.

5. FASTER ALGORITHMS FOR SEQUENTIAL SEARCH

In comparing the speed with which the Terasys and a conventional computer can perform nearest neighbor search, we will assume that both the conventional computer and the Terasys use exhaustive search to find nearest neighbors. Consequently, we must consider whether there are sequential nearest neighbor search techniques which are more efficient than exhaustive search.

Euclidean search is a special case of nearest neighbor search where records correspond to points in a Euclidean space. As discussed in Appendix A, Euclidean search has been the subject of a substantial amount of research.

The execution time of exhaustive search is linear in the size of the search space. A goal of research into Euclidean search is to find search algorithms with sublinear performance. Unfortunately, the sublinear Euclidean search algorithms that have been developed so far tend to work well only in low-dimensional spaces.

The performance of sublinear search algorithms in practice strongly depends on the distribution of points in the search space. When the distribution of points is fairly uniform, the performance of sublinear search algorithms tends to follow the following pattern. The algorithms typically do well in cases where k (the dimension of the space) is fairly small, i.e., less than 8. For values of k between about 8 and about 20, the application of sublinear techniques tends to become increasingly difficult for one of two reasons: either performance degenerates into exhaustive search or the number of data points required for efficient operation becomes unrealistically large. For values of k much larger than about 20, the two problems just described tend to become so severe that using existing techniques to achieve a significant speedup over exhaustive search becomes essentially impossible for data sets of a reasonable size. The problem of finding efficient Euclidean search techniques for high-dimensional spaces is an active area of research.

The extent to which the Euclidean search methods described in Appendix A can be applied to the more general kinds of nearest neighbor search will vary from case to case. If we permit the distance between two records to be defined by an arbitrary function, it seems that achieving a guaranteed speedup over exhaustive search is not possible even in principle. In practice, it may be reasonable to assume that the function f defining the distance between two records is always a *metric,* meaning that f has the following three properties:

1. $f(R_1, R_2) \geq 0$ for all R_1 and R_2
2. $f(R_1, R_2) = 0$ only if $R_1 = R_2$
3. $f(R_1, R_3) \leq f(R_1, R_2) + f(R_2, R_3)$.

A nearest neighbor search technique discussed in [Shas 1990] can be applied to any search problem where a metric distance function is used. We would expect, however, that like the Euclidean search methods described in Appendix A, the method described in [Shas 1990] will become inefficient for records with many parameters (i.e., high-dimensional records).

As will be discussed in Section 8, one area where a Terasys implementation of nearest neighbor search may be particularly useful is during the development of new nearest neighbor classifiers. During classifier development, it may be necessary to experiment with different distance functions and to alter other parameters of the search as well. In such a context, it may be especially difficult to implement sequential search techniques with better performance than exhaustive search.

6. THE EFFECT OF I/O ON TERASYS PERFORMANCE

Given two records R_1 and R_2, we refer to the process of computing the distance between R_1 and R_2 as a *record comparison operation.* Assuming that R_1 is stored in host memory and that R_2 is stored in the memory of a single Terasys processor, let s_1 be the number of record comparison operations per second that a single Terasys processor can perform. Let s_2 be the number of record comparison operations per second that can be performed by a specific conventional computer. For a given type of record and a given distance measure, we refer to the quantity $r = s_1/s_2$ as the *performance ratio* of a single Terasys processor with respect to the conventional computer. Since a single Terasys processor will typically be much slower than a conventional computer, quantity r will normally be smaller than 1.

For a Terasys workstation with P processors, the maximum speedup that the Terasys can achieve with respect to a given conventional computer is generally equal to Pr. In practice, the actual speedup achieved by a Terasys workstation will be less than Pr because of the time that must be spent on I/O.

When we compare a Terasys workstation with a conventional computer, we will assume that the conventional computer and the Terasys host have the same amount of primary memory and the same secondary to primary transfer rate. Under these assumptions, both a conventional and Terasys implementation of a particular algorithm will spend the same amount of time doing secondary to primary transfer.

Amdahl's law [Marc 1994, p. 775] can be applied to the Terasys as follows. Suppose that a given program requires t seconds to run on a sequential computer and that of those t seconds, h seconds are devoted to tasks which must be run on the Terasys host. In such a case, a Terasys workstation must take at least h seconds to perform the task. Consequently, the maximum speedup that the Terasys can achieve is t/h.

For nearest neighbor search, the most significant part of the problem that must be performed on the host is the transfer of records from secondary to host memory. Suppose that for a given search problem the conventional computer spends ninety percent of its total computation time on secondary to primary transfer. In this case, the maximum speedup the Terasys can achieve is 10/9 or approximately ten percent.

For a search problem to be a good candidate for the Terasys, we might want to make sure that a conventional implementation of the problem spends no more than one or two percent of its total computation time on secondary to primary transfer. As illustrated by the examples in the following section, for large sets of records such low percentages should not be difficult to achieve.

7. EXPERIMENTAL RESULTS

The Terasys used in the following tests had 32K processors and was connected to a Sun Microsystems Sparcstation-2 host. The conventional workstation used for comparison was a 61 MHz Sun Microsystems Sparcstation-20. The records used in the experiment were vectors of 32 8-bit positive integers. The distance between two vectors was defined to be the square of the Euclidean distance.

On the Sparcstation-20 that we tested, it turned out that floating point arithmetic was faster than integer arithmetic. Consequently, in a Sparcstation-20 implementation of nearest neighbor search it would be advantageous to store vector components on disk as 8-bit integers but to convert vector components to floating point values before processing begins.

Our analysis of conventional workstation performance is based on the following experimental results:

- A Sparcstation-20 can transfer approximately 92,600 vectors per second from disk to host memory. (Note: All of our experimental results are rounded to three decimal places of accuracy.)

  Since each vector is 32 bytes long, 92,600 vector transfers per second corresponds to approximately 2,960,000 bytes per second.
- A Sparcstation-20 can convert approximately 75,200 vectors per second from integer to floating point components.
- A Sparcstation-20 can perform approximately 158,000 vector comparisons per second.

  Given two vectors *U* and *V*, computing $(U_i - V_i)^2$ for a single component requires 2 floating point operations. Computing $(U_i - V_i)^2$ for 32 components requires 64 floating point operations. Computing the sum of $(U_i - V_i)^2$ over 32 components requires an additional 31 floating point operations. Computing the square of the distance between *U* and *V* thus requires 95 floating point operations. Consequently, 158,000 vector comparisons per second corresponds (very roughly) to 15,000,000 floating point operations per second.

Our analysis of Terasys performance is based on the following experimental results:

- A Terasys workstation can transfer approximately 3,850 vectors per second from host memory to the Terasys processor array.

  A transfer rate of 3,980 vectors per second corresponds 123,000 bytes per second. The reason the transfer rate is so slow is that we used dbC assignment statements to write individual vectors to individual processors. Using this approach, essentially only a single bit was transferred to a single processor during each cycle. As discussed earlier in the report, the time required to load the Terasys array could have been substantially reduced if we had stored the data in the host in corner-turned format.

- A Terasys workstation with 32K processors can perform approximately 10,900,000 vector comparisons per second.

  For the Terasys, each vector comparison requires 95 small integer operations. Consequently, 10,900,000 vector comparisons per second roughly corresponds to slightly more than 1 billion small integer operations per second.

7.1 Performance Ratio of the Terasys

The experimental results given in the previous section indicate that a Sparcstation-20 can perform approximately 158,000 vector comparisons per second while a 32K processor Terasys can perform approximately 10,900,000 vector comparisons per second. This means that a 32K Terasys is able to compare vectors approximately 69 times faster than a Sparcstation-20. Our experimental results also imply that a single Terasys processor can perform approximately 333 vector comparisons per second. This means that a Sparcstation-20 is about 474 times as fast as a single Terasys processor.

7.2 Estimates of Overall Terasys Performance

In the previous section, we noted that a 32K processor Terasys can compare vectors approximately 69 times faster than Sparcstation-20. As discussed earlier, a factor of 69 is thus an upper bound on the overall speedup that the Terasys can achieve for this problem. In the following two examples, we estimate the total execution times of a Sparcstation-20 and Terasys for vector sets of different sizes. (Note: As we proceed through our examples, we will immediately round each result to three decimal places of accuracy.)

Example: Find the nearest neighbors of 10,000 sample vectors in a training set of 100,000 vectors.

1. Estimate of I/O time for Sparcstation-20

- Time to transfer 10^5 training vectors from host to disk: 1.08 seconds.
- Time to convert 10^5 training vectors from integer to floating point: 1.33 seconds.
- Time to transfer 10^4 sample vectors from host to disk: 0.108 seconds.
- Time to convert 10^4 sample vectors from integer to floating point: 0.133 seconds.
- Total I/O time: 2.65 seconds.

2. Estimate of total execution time for Sparcstation-20

- Time to perform $10^4 * 10^5$ comparisons: 6,330 seconds.
- Total Sparcstation-20 execution time: 6,330 seconds = 106 minutes. (I/O time is insignificant.)

3. Estimate of I/O time for Terasys

- Time to transfer 10^5 training vectors from host to disk: 1.08 seconds.
- Time to transfer 10^4 sample vectors from host to disk: 0.108 seconds.
- Time to transfer 10^5 training vectors from host to Terasys array: 26.0 seconds.
- Total I/O time: 27.2 seconds.

4. Estimate total execution time for Terasys

- Time to perform $10^4 * 10^5$ comparisons: 91.7 seconds.
- Total Terasys execution time: 91.7 + 27.2 = 119 seconds.
- **Overall Terasys speedup: 6,330/119 = 53**

Example: Find the nearest neighbors of 100,000 sample vectors in a training set of 1,000,000 vectors.

1. Estimate of I/O time for Sparcstation-20

- Time to transfer 10^6 training vectors from host to disk: 10.8 seconds.
- Time to convert 10^6 training vectors from integer to floating point: 13.3 seconds.

- Time to transfer 10^5 sample vectors from host to disk: 1.08 seconds.
- Time to convert 10^5 sample vectors from integer to floating point: 1.33 seconds.
- Total I/O time: 26.5 seconds.

2. Estimate of total execution time for Sparcstation-20

- Time to perform $10^5 * 10^6$ comparisons: 633,000 seconds.
- Total Sparcstation-20 execution time: 633,000 seconds = 10,600 minutes = 177 hours = 7.38 days. (I/O time is insignificant.)

3. Estimate of I/O time for Terasys

- Time to transfer 10^6 training vectors from host to disk: 10.8 seconds.
- Time to transfer 10^5 sample vectors from host to disk: 1.08 seconds.
- Time to transfer 10^6 training vectors from host to Terasys array: 260 seconds.
- Total I/O time: 272 seconds.

4. Estimate of total execution time for Terasys

- Time to perform $10^5 * 10^6$ comparisons: 9,170 seconds.
- Total Terasys execution time: 9,170 + 272 = 9,440 seconds = 157 minutes = 2 hours, 37 minutes.
- **Overall Terasys speedup: 633,000/9,440 = 67**

8. USING THE TERASYS DURING CLASSIFIER DEVELOPMENT

One area where a Terasys implementation of nearest neighbor search may be particularly useful is in the development of new nearest neighbor classifiers. During the development of a new classifier, it may be necessary to experiment with a number of different classifier designs. For example, a user might want to test different methods of extracting features from the training data or try out different distance functions. During the development process, the ability to test a classifier in less time means that more tests can be conducted and that more variations can be tried. The ability to test a classifier quickly is especially important when a technique such as genetic algorithms is used to automatically generate classifier parameters.

Because the time required to load training vectors must be amortized over many comparisons, a Terasys implementation of nearest neighbor search can usually be applied effectively only when many samples need to be processed simultaneously. The classifier test and refinement techniques discussed in this section both meet this criterion.

8.1 Estimating the Error Rate of a Nearest Neighbor Classifier

Consider the problem of estimating the error rate of a nearest neighbor classifier which classifies samples using a training set T. As discussed in [Fuku 1990, p. 219], one technique for estimating the error rate of the classifier is to determine how well a subset of the records in T is classified by the remaining records. In the *leave-one-out* error estimation method, a single record is removed from T and then classified using the remaining records. The process is repeated until enough of the records in T have been classified to yield a reliable error estimate.

With only minor changes, the Terasys nearest neighbor search algorithm that we have discussed can be used to implement the leave-one-out method. Suppose that it has been determined that a total of n records need to be tested in order to reliably estimate the error rate of a classifier. A Terasys implementation of the leave-one-out method would generate a sample set by randomly selecting n records from T. The n records in the sample set would then be classified using the entire training set minus the sample currently being classified.

8.2 Refining the Training Set

Given a nearest neighbor classifier based on a training set *T*, one would expect that deleting a subset of the records in *T* would improve the speed of the classifier at a cost of increasing its error rate. In such a situation, the question arises of how to choose a subset of records to delete in a way that minimizes the resulting increase in errors.

A substantial amount of research in the area of training set refinement has been done. In [Dasa 1991], 18 papers proposing a variety of refinement techniques were reviewed. In this section, we propose a training set refinement technique that is well-suited for implementation on the Terasys.

The training set refinement technique that we propose repeatedly identifies and removes the least useful record in the original training set. The method we describe is closely related to the *reduced nearest neighbor (RNN) rule* [Gate 1972]. The RNN rule begins the process of removing redundant records only after the original training set has been refined using another method called the *condensed nearest neighbor rule* [Hart 1968]. In contrast, our refinement technique is applied directly to the original training set. Both the condensed nearest neighbor rule and the RNN rule assume that only a single nearest neighbor is used to classify each sample. The method we describe can be used to refine classifiers which make use of several nearest neighbors.

Given a training set *T*, our refinement method starts by using the leave-one-out technique described in the previous section to estimate the error rate of *T*. Suppose it has been determined that a total of *n* records from *T* need to be classified in order to obtain a reliable error estimate. Our algorithm begins by building a sample set *S* consisting of *n* randomly chosen records from *T*. An initial estimate of *T*'s error rate is then obtained by classifying each record in *S* using the leave-one-out method. Once the initial error estimate has been computed, our refinement procedure proceeds to identify records in *T* whose deletion would cause little or no increase to the initial error rate.

Suppose the records in *T* are used to classify a sample according to its *m* nearest neighbors. During the second stage, our method uses the Terasys to find and store a list of at least the *m* nearest neighbors of each sample in *S*. Up to some limit, it will be beneficial to store as many neighbors for each sample in *S* as possible.

Once a list of nearest neighbors for each record in *S* has been computed, the third stage of our refinement process starts. We assume that the third stage is performed entirely on the host computer but it is interesting to note that under some circumstances the Terasys could be used for the third stage as well.

During the third stage, each record *U* in *T* is examined to determine how the deletion of *U* would change the error rate of the current training set. To determine how the deletion of *U* would affect the classification of a particular sample record *R*, the list of *R*'s *m* nearest neighbors is considered. If *U* is not one of the *m* nearest neighbors of *R*, deleting *U* would clearly not affect the classification of *R*. If *U* is one of the *m* nearest neighbors of R, the deletion of *U* might change *R*'s classification from correct to incorrect, might change *R*'s classification from incorrect to correct, or might leave *R*'s classification unchanged.

After the overall effects of deleting each training record have been considered, the training record which would decrease the error rate the most or increase the error rate the least is deleted. This process is repeated until any further deletions would result in an error rate that exceeds some threshold.

As more and more training records are deleted, the lists of active nearest neighbors for each of the sample records will become shorter and shorter. At some point they will be so short that the *m* nearest neighbors for one or more samples can no longer be determined. When this happens, the Terasys is used again to "refresh" the nearest neighbor lists for all samples with missing elements.

9. ORGANIZING THE TERASYS INTO VECTOR COMPARISON UNITS

Up until now, we have assumed that nearest neighbor search is implemented on the Terasys by placing one record in each processor. In a case where a each record is a vector of similar components, it is sometimes possible to implement nearest neighbor search more efficiently on the Terasys by placing vectors linearly along the Terasys array with adjacent vector components stored in adjacent processors. For this approach to be possible, the distance function between two vectors must first apply the same operation to each pair of components and then apply a global operation such computing a sum or count to the results. We will illustrate this technique for the case where records correspond to points in a Euclidean space. Other types of distance functions where the "one-component-per-processor" approach could be applied include:

- A distance function that counts the number of pairs of components that have identical values.
- A distance function that computes the average difference between pairs of components.
- A distance function that computes the maximum difference between pairs of components.

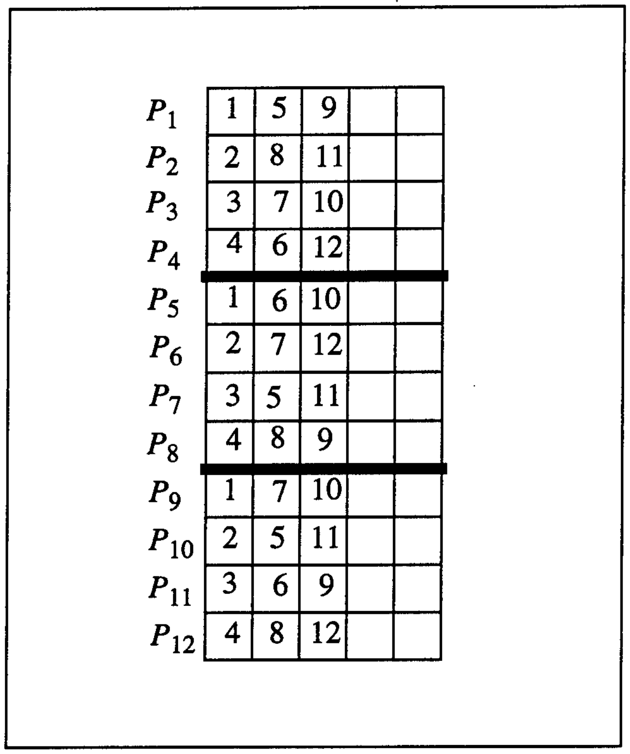
9.1 Using Vector Comparison Units to Compute Euclidean Distances

Consider the case where each record denotes a point in a k-dimensional Euclidean space. We will assume that each point in the search space is represented by a vector of k integers where k is a power of 2. The one-component-per-processor approach divides the processor array into segments of k processors and uses each segment to store one sample vector and a fixed number of training vectors . We refer to each segment of k processors as a *vector comparison unit* or *VCU.*

Storing vectors in VCUs makes it possible to take advantage of the special Terasys hardware discussed in Section 3 for efficiently finding the sum of variables in a segment of processors. The Euclidean distance *e* between two vectors *U* and *V* is defined as

$$e = \sqrt{(U_1 - V_1)^2 + \ldots + (U_k - V_k)^2}.$$

In the case where both *U* and *V* are stored in the same VCU, each processor in the VCU can compute the quantity $(U_i - V_i)^2$ for the components of *U* and *V* stored in the processor. Once these quantities have been computed, the Terasys summation hardware is used to compute their sum in time *O(log k)* instead of time *O(k)*. Because the squares of nonnegative integers increase monotonically, in a Euclidean search problem quantity e^2 can be used for comparisons in place of quantity *e*.



P_1	1	5	9		
P_2	2	8	11		
P_3	3	7	10		
P_4	4	6	12		
P_5	1	6	10		
P_6	2	7	12		
P_7	3	5	11		
P_8	4	8	9		
P_9	1	7	10		
P_{10}	2	5	11		
P_{11}	3	6	9		
P_{12}	4	8	12		

Figure 1. Sample use of VCUs

Example: Figure 1 illustrates how VCU-based search might be implemented in a case where vectors have 4 components. In this example, we assume that the Terasys processor array contains just 12 processors and that each processor has 5 bytes of memory. In figure 1, the array of 12 processors has been divided into 3 VCUs. The first column of each VCU holds the current sample vector--in this case [1, 2, 3, 4]. Columns 2 and 3 hold training vectors, and columns 4 and 5 are reserved for intermediate processing. During the first cycle of the algorithm, sample vector [1, 2, 3, 4] is compared with training vectors [5, 8, 7, 6], [6, 7, 5, 8], and [7, 5, 6, 8] in parallel. During the second cycle, vector [1, 2, 3, 4] is compared with vectors [9, 11, 10, 12], [10, 12, 11, 9], and [10, 11, 9, 12].

Example: As a second example, we consider a VCU-based search implementation in a more realistic case where each vector has 256 8-bit components and where the Terasys processor array contains 32K processors. In this case, we divide the 32K Terasys processor array into 128 vector comparison units, each consisting of 256 processors. If each VCU stores one sample vector and 249 training vectors, the *i*th processor in each VCU will store the *i*th component of the sample vector and the *i*th component of each of the 249 training vectors. The number of bits required per processor is thus 250 * 8 = 2,000. Since each processor has 2,048 bits of local memory, this will leave 48 bits free per processor for use in intermediate processing.

9.2 Optimal Terasys Design for VCU-Based Search

We have already pointed out that one advantage of using VCUs to implement nearest neighbor search is that special Terasys hardware for computing the sums of segments can be used. A second advantage of using VCUs is that VCU-based search can efficiently handle vectors of length equal to any power of 2.

Trying to store an entire vector in each processor can be inefficient whenever vectors are either too big or too small to fit into a single processor. Consider, for example, the case where a single vector must be stored in two processors. In such a case, the two processors used to store a given vector must do their processing sequentially rather than in parallel. Such an implementation is inefficient because half of the processors are idle during any given processing step.

As discussed earlier, a single Terasys chip contains 64 processors. The case where vectors are too small to fill up a single processor's memory is also inefficient because in such a case space on the Terasys chip that could have been used for more processors has instead been used for unnecessary memory space.

It seems that for VCU-based search, an optimal Terasys design would have as many processors as possible per chip. The memory of each processor should be just large enough to store two vector components with enough intermediate processing space left over to compute a simple quantity like $(U_i - V_i)^2$. In this design, each VCU would store just one sample vector and one training vector. A Terasys designed in this way would be able to efficiently perform VCU-based search for vectors of length k where k is any power of 2 as long as the Terasys contains substantially more than k processors. The reason why the Terasys must contain substantially more than k processors is as follows.

In the optimal Terasys design just described, a different training vector is loaded into each VCU. We will assume that the cost of loading training vectors into VCUs can be amortized by comparing the loaded training vectors with many sample vectors.

In order to compare a single sample vector with each of the loaded training vectors, the sample vector must be broadcast one time to load the sample vector into each VCU. When there are many VCUs, the time spent broadcasting samples will have little effect on overall performance. If there are only a few VCUs, the time spent broadcasting samples could significantly degrade performance. In the extreme case where the entire processor array is configured as a single VCU containing just a single training vector, broadcasting a single sample vector is likely to take more time than a sequential computer would require to compare the sample with the training vector.

10. PATTERN RECOGNITION APPLICATIONS

In this section we consider how the Terasys could be used to search for patterns in time series data and in 2-dimensional images. The fact that Terasys processors are arranged in a linear array makes the Terasys particularly well-suited for the task of time series analysis.

10.1 Time Series Analysis

Consider, as an example, the problem of continuously monitoring a patient's EKG. Assume that an EKG signal can be converted into a time series of integers and that a training set *T* exists where each vector in *T* corresponds to a potentially significant pattern. For the sake of simplicity, assume the vectors in *T* are all of length *k* where *k* is a power of 2.

Assume that our goal is to design a classifier which will post an alert each time a length-*k* subsequence of the input time series falls within a fixed distance of a vector in *T*. VCU-based search can be used to solve this problem by dividing the Terasys into vector comparison units each of length *k*. As before, each VCU stores a fixed number of training sequences. The input time series is fed one value at a time through the processor array. Each time the input time series is advanced one step, a new length-*k* subsequence is, in effect, loaded into each VCU. Each VCU then computes the distance between its current subsequence and each of the training sequences that it contains.

In a case where the entire training set can be stored in Terasys memory, the implementation that we have just described could potentially analyze input signals in real time. In a case where the training set is too large to fit in Terasys memory, the training set could be broken into subsets. In this case it would probably be necessary to store the input signal and analyze it off-line. Each time a new subset of training sequences was loaded, the entire input signal would be fed through the processor array.

10.2 Image Analysis

In principle, nearest neighbor search can be directly applied to image analysis. Consider a target recognition system which is responsible for locating targets in an image. Assume that the system makes use of a set of training images in which each target has been photographed at all scales and orientations likely to appear in samples.

Assume that each input image is of size *m* by *m* and that each target image is of size *n* by *n* where *n* is smaller than *m*. Each *m* by *m* input image can be viewed as a set of overlapping *n* by *n* subimages. By viewing each *n* by *n* subimage as a linear vector of length n^2, target recognition can be converted into nearest neighbor search. A possible distance measure in this case might be the total number of target pixels that closely match the corresponding pixel in an input subimage.

In most cases, the number of training vectors required to cover every possible scale and orientation of each target is likely to be unrealistically large. A more traditional approach to target recognition might initially use a conventional computer to segment an input image into separate objects. Each object would then be processed to extract a set of features for the object. Assuming that the target images have been processed in the same way, nearest neighbor search could then be used to classify the resulting feature vectors.

11. OTHER APPLICATIONS OF EUCLIDEAN SEARCH

In this section we consider several applications of Euclidean search which are not directly related to classification. We consider how Euclidean search can be applied to the problems of estimating intrinsic dimensionality and computing the approximate value of a density function. We also consider how density estimation based on Euclidean search can be used to generate new vectors which have approximately the same distribution as the vectors in an existing set.

11.1 Estimating Intrinsic Dimensionality

Consider a random vector X of the form $[x_1, \ldots, x_k]$ where each x_i is a random variable. The explicit dimensionality of X is equal to k. As discussed in [Fuku 1990, p. 280], we say that the *intrinsic* or *effective* dimensionality of X is equal to k_e if each component x_i of X can be written as a function $f_i(y_1, \ldots, y_{k_e})$ of k_e random variables. Geometrically, this means that each sample of X lies on a hypersurface of topological dimension k_e.

Example: Consider a set of points which are randomly distributed on a 2-dimensional plane embedded in a 10-dimensional space. In this case, the explicit dimensionality of the points is 10 while their intrinsic dimensionality is 2.

Knowing the intrinsic dimensionality of a data set can be useful in classification problems. If the intrinsic dimensionality of a set of points is significantly lower than its explicit dimensionality, it may be possible to map the points into a lower dimensional space in which the classification task can be performed more efficiently.

In [Fuku 1990, p. 281], the following formula is given which can be used to estimate the intrinsic dimensionality of a set of data points:

$$\frac{E\{d_{(m+1)NN}(X)\}}{E\{d_{mNN}(X)\}} \cong 1 + \frac{1}{mk_e}.$$

In this expression, quantity $E\{\,d_{(m+1)NN}(X)\,\}$ is the expectation of the distance from a sample of X to its $(m + 1)$th nearest neighbor. Quantity $E\{\,d_{mNN}(X)\,\}$ is the expectation of the distance from a sample of X to its mth nearest neighbor.

Given a set of samples, the two expectation values in the above formula can be estimated by choosing an arbitrary value of m and computing the average distance to the mth and $(m + 1)$th nearest neighbor of each sample. Once estimates of $E\{\,d_{(m+1)NN}(X)\,\}$ and $E\{\,d_{mNN}(X)\,\}$ have been computed, an estimate of the intrinsic dimensionality k_e of the sample set can be obtained using the equation given earlier.

11.2 A Nonparametric Technique for Estimating Density

Suppose the only information available about a random variable X is a set S of samples of X. Let p be the (unknown) probability density function associated with X and suppose we want to evaluate p at an arbitrary point V. As discussed in [Fuku 1990, p. 255], one technique for evaluating $p(V)$ is to center a unit volume kernel function at each of V's m nearest neighbors in S. The sum of the kernel functions at V is then used as an estimate of $p(V)$. We refer the reader to [Fuku 1990] for a detailed description of this method. In a case where the density function p needs to be evaluated at many points, the Terasys nearest neighbor search algorithm discussed in this report could be used to find the m nearest neighbors of each point.

One interesting application of nonparametric density estimation is in the generation of new samples that have approximately the same distribution as the vectors in an existing set. As before, let S be a set of samples generated by a random variable X and let p be the unknown probability density function associated with X. Suppose we want to generate new samples of X which lie within a k-dimensional region R. Although the structure of p is unknown, suppose it can be assumed that p is bounded by a constant c throughout R.

The *rejection method* [Pres 1992, p. 290] for generating new samples of X can be applied in our example as follows. First, generate a random point V in R and a random number r between 0 and c. The density estimation technique described earlier in this section is then used to obtain an estimate of $p(V)$. If r is less than the estimate of $p(V)$, V is returned as the new sample of X. If r is greater than the estimate of $p(V)$, the process is repeated.

Recall that the Terasys can be used effectively for nearest neighbor search only in cases where the nearest neighbors of many points need to be found simultaneously. When the rejection method is used to generate new samples, the generation of a single new sample can require multiple evaluations of p which means that the nearest neighbors of many points need to be found. The fact the nearest neighbors of many points need to be found means that Terasys-based nearest neighbor search can potentially be applied even when only a few new samples need to be generated.

12. APPLICATIONS OF EUCLIDEAN SEARCH TO RADAR ANALYSIS

To illustrate how the Terasys could be applied to the kinds of government work conducted by IDA divisions outside of the Center for Computing Sciences, we examined three possible applications of a Terasys implementation of Euclidean search to an ongoing radar analysis project being conducted by Jim Silk in IDA's Science and Technology Division. To find out how much computing power these applications require, we built prototypes of the three applications using a conventional workstation to implement Euclidean search. All the experiments discussed in this section were conducted using a 61 MHz Sun Microsystems Sparcstation-20.

All three of our tests made use of a set of tank radar data which had been previously obtained in a laboratory. To collect the data, a tank was mounted on a turntable and rotated through a little less than 360 degrees in a total of 29,516 increments. At each increment, a radar scan of the tank was obtained. Before the data was used in our experiments, each scan was converted into a vector of 32 real-valued components where each component was normalized to lie between 0 and 1.

This data set of approximately 30,000 vectors was used only for demonstration purposes. To fully analyze an actual radar system, a data set containing substantially more than 30,000 vectors would be required.

12.1 Target/Clutter Discrimination

The first test that we performed used nearest neighbor search to classify samples as either targets or randomly generated "clutter" based on the classifications of their nearest neighbors. In this experiment our training set consisted of 59,032 vectors, half of which were actual tank scans and half of which were randomly generated. Our sample set consisted of 6,000 vectors, half of which were actual tank scans and half of which were randomly generated. Our experiment placed each sample vector into the class of its single nearest neighbor. (Since each tank scan sample vector was also a training vector, tank scan samples were classified according to their second nearest neighbors.)

In the experiment, the Sparcstation-20 was able to classify the entire sample set in 37.6 minutes, meaning that approximately 2.7 vectors were classified per second. The misclassification rate for both the target and random vectors was about 4 percent.

In this example, if the 6,000 samples vectors needed to be classified just one time, having to wait 37 minutes may not be a significant problem. If one wanted to improve the accuracy of the classifier, however, having to wait more than a half hour to see the results of a modification could be a serious impediment.

As discussed earlier, an actual radar evaluation experiment would be likely to use substantially more than 30,000 (or 60,000) training vectors. The speedup that the Terasys could provide might become very significant if the sizes of both the target and training sets were increased by a factor of 10. In such a case, the execution time for the Sparcstation would grow by a factor of 100 to more than 50 hours.

12.2 Effect of Conversion to Integers on Classifier Accuracy

In the classification experiment discussed in the previous section, each tank-scan vector component was normalized to a floating point value between 0 and 1. In order to perform vector search on the Terasys, each vector component would probably need to be converted to a small integer. We performed the following test (using a Sparcstation) to analyze the effect of conversion to 8-bit integers.

The real-valued vectors used in our conversion test consisted of the 29,516 tank-scan vectors combined with 29,516 randomly generated vectors. A corresponding integer-valued training set was constructed by converting each vector component to a value between 0 and 255. The first sample set consisted of 10,000 randomly chosen tank-scan vectors and the second sample set consisted of 10,000 random vectors. Integer-valued versions of both sample sets were constructed.

When the tank-scan sample set was tested, approximately one vector in 74 had a different nearest neighbor in the integer-valued training set than the corresponding vector had in the real-valued training set. For one tank-scan sample in 1,250, the nearest neighbor in the integer-valued training set was in a different class than the nearest neighbor in the real-valued set.

When the random sample set was tested, approximately one vector in 60 had a different nearest neighbor in the integer-valued training set than the corresponding vector had in the real-valued training set. For one random sample in 588, the nearest neighbor in the integer-valued training set was in a different class than the nearest neighbor in the real-valued set.

These results imply that for this particular classification problem, conversion from real-valued to 8-bit components has only a small effect on classifier accuracy.

12.3 Dimensionality Test

The explicit dimensionality of the vectors used in our tests was 32. Using the technique described in Section 11.1, we used the average distances to nearest neighbors to estimate the intrinsic dimensionality of the tank scan data.

Our training set in this case consisted of the 29,516 tank scan vectors and our sample set consisted of 3,000 vectors randomly chosen from the training set. Our test computed the average distances to the 20 closest neighbors of 3,000 sample points. Finding the 20 closest neighbors of 3,000 samples required approximately 26 minutes of Sparcstation-20 computing time. The initial result of the dimensionality test indicated that dimensionality of the tank scan data set is about 15.

If the intrinsic dimensionality of a data set is k_e, the dimensionality test described in Section 11.1 can be applied directly only if the data set is large enough to "fill" a space of dimension k_e. Calibration tests showed that the relatively small size of the tank scan data set was skewing the calculated intrinsic dimensionality to an artificially small value. We ran the dimensionality test program a number of times to test different techniques to compensate for this bias.

A conventional workstation was fast enough to compute the intrinsic dimensionality of a data set of about 30,000 points. As in the previous example, in a case where the sample and training sets were 10 times larger and the execution time was 100 times larger, the Terasys could provide a significant advantage.

12.4 Entropy Computation

Assume that the 29,516 vectors in the tank-scan data set are samples of a random vector X and let p be the probability density function associated with X. The entropy e associated with p is defined as

$$e = \int_{-\infty}^{\infty} p(U) \log p(U) \; dU .$$

Monte Carlo integration [Pres 1992, p. 304] is a technique for integrating a function by evaluating the function at a large number of random points. For our data set, we found that evaluating the quantity $p(U) \log p(U)$ at 1,000 random points was sufficient to yield a stable integral. At each point, we used the nonparametric density estimation method described in Section 11.2

to evaluate $p(U)$. The entire process of computing the entropy of the tank scan data set required approximately 9.7 minutes of computing time on a Sparcstation-20.

As in our other examples, for a data set of about 30,000 vectors a conventional workstation was fast enough to perform the entropy computation. For a larger data set, the Terasys could potentially provide a significant advantage.

13. CONCLUSIONS AND RECOMMENDATIONS

This report has demonstrated that nearest neighbor search can be implemented efficiently on the Terasys and that nearest neighbor search can be applied to a number of problems that are potentially of interest to IDA sponsors.

One area where the use of Terasys-based nearest neighbor search seems particularly promising is in the development of new nearest neighbor classifiers. During classifier development, the ability to test a classifier in less time means that more tests can be conducted and that more variations can be tried. The ability to test a classifier quickly is especially important when a technique such as genetic algorithms is used to automatically generate classifier parameters.

Euclidean search is a special case of nearest neighbor search where records corresponds to points in a Euclidean space. In the report we have discussed how Euclidean search can be efficiently implemented on the Terasys by dividing the Terasys array into vector comparison units. We have considered a number of areas where Euclidean search can be applied including several applications in the area of radar analysis.

We recommend that the next step in this investigation should be additional experiments which use Terasys-based nearest neighbor search in a variety of applications. We particularly recommend tests that would use the Terasys to assist in the development of new nearest neighbor classifiers.

APPENDIX A.
IMPROVING THE EFFICIENCY OF EUCLIDEAN NEAREST NEIGHBOR SEARCH

Let T be a set of points in a k-dimensional Euclidean space and consider the problem of finding the nearest neighbor in T of an arbitrary point V. If there are N points in T, comparing V with every point in T will take time $O(N)$. In this appendix we consider several techniques which can potentially find the nearest neighbor of V without examining every point in T.

For three of the techniques we discuss, some sort of average or worst case sublinear performance bound is provided. Unfortunately, the three techniques with guaranteed sublinear performance all tend to become impractical in high-dimensional spaces.

The fourth technique we consider, a method using Fourier coefficients as indices, is heuristic in the sense that no guarantee of sublinear performance is provided. The ability of heuristic techniques to perform Euclidean search will vary from application to application.

A.1 The Friedman, Baskett, and Shustek Algorithm

One of the first Euclidean search algorithms with provably sublinear expected execution time was developed by Friedman, Baskett, and Shustek [Frie 1975]. We refer to this as the FBS algorithm. Like most of the algorithms discussed in this appendix, the FBS algorithm can be used to find the m nearest neighbors of a sample point. For simplicity, we consider only the case where $m = 1$.

The FBS algorithm requires time $O(N \log N)$ to preprocess the points in T and is able to find the nearest neighbor of an arbitrary sample point in expected time $O(N^{1-1/k})$. In small dimensional spaces, the FBS algorithm can achieve a significant speedup for data sets of a reasonable size. Consider, for example, the case where $k = 2$. In this case the time required for the FBS algorithm to find the nearest neighbor of an arbitrary point is $O(\sqrt{N})$. Ignoring constant terms, in the 2-dimensional case T might need to contain only 100 points in order for the FBS algorithm to achieve a ten-fold speedup over exhaustive search.

For higher dimensions, however, the number of points required for the FBS algorithm to obtain a significant speedup can become extremely large. In the case where $k = 32$, the time required by the FBS algorithm is approximately $O(N^{0.969})$. In this case (again ignoring constant terms), T might need to contain approximately $1.8 * 10^{32}$ points in order for a ten-fold speedup over exhaustive search to be achieved.

A.2 Cell Methods

Dividing a 2-dimensional search space into a grid of 2-dimensional cells is often an effective means of performing nearest neighbor search. Consider the problem of searching for nearest neighbors within a 2-dimensional square. In this case, a grid of n by n cells could be implemented using an n by n array. Each location in the array points to a linked list of all the points which lie within the corresponding cell. The nearest neighbor of an arbitrary point V is found by searching in a spiral pattern beginning with the cell containing V.

Conceptually, the 2-dimensional search technique just described can be generalized to higher dimensions in a straightforward way. A k-dimensional space can be divided into hypercubic cells each of dimension k. The search for the nearest neighbor of a point V begins in the cell where V is located and proceeds to the neighbors of that cell. Bentley et al. [Bent 1980] analyze the performance of this approach. They show that for any value of k, if the points in T are uniformly distributed in a hypercube of dimension k, then the average number of cells that need to be searched to find the nearest neighbor of an arbitrary sample point is $O(1)$.

There are several techniques which can be used to implement a multidimensional cell array using a reasonable amount of space. By hashing the indices of active cells, a hash table can be used to implement the cell array in space $O(N)$ with potentially constant access time. A variation of the *k-d* tree described later in this appendix can be used to implement the cell array in space $O(N)$ with access time $O(log\ N)$.

Unfortunately, cell methods encounter a serious obstacle when generalized to a high-dimensional space. As discussed in [Bent 1980], each cell in a k-dimensional space has $3^k - 1$ neighbors. Consequently, even though the average number of cells that need to be searched is $O(1)$, the constant of proportionality grows exponentially with the dimension of the search space. Because of this exponential dependence, cell-based techniques are likely to become impractical in high-dimensional spaces.

A.3 Approximate Nearest Neighbor Search

As we have just discussed, Bentley et al. [Bent 1980] show that for a uniform distribution of points the average number of cells that need to be searched to find the nearest neighbor of an arbitrary sample point is $O(1)$. Recently, this result has been strengthened in the following way [Arya 1993, 1994].

Consider, as before, a set T of N points in a k-dimensional space. Let V be an arbitrary sample point. Following [Arya 1994], we define an approximate nearest neighbor of V as follows. For any $\varepsilon > 0$, we say that a point U in T is a $(1 + \varepsilon)$-*nearest neighbor* of V if for all points U' in T

$$\frac{dist(V, U')}{dist(V, U)} \leq 1 + \varepsilon .$$

In the case where $\varepsilon = 0.1$, the set of $(1 + \varepsilon)$ nearest neighbors of V consists of all points which are up to ten percent further from V than the actual nearest neighbor of V.

Arya et al. present a cell-based Euclidean search technique for which they are able to give worst case rather than average case performance results. Using a tree-based structure to implement a multidimensional cell array, they show that for any distribution of points and for any non-zero value ε, at most $O(1)$ cells need to be examined to find a $(1 + \varepsilon)$-nearest neighbor of an arbitrary sample point. In their method, the time required to find the cell containing an arbitrary point is $O(log\ N)$. Consequently, their method is able to find a $(1 + \varepsilon)$-nearest neighbor of an arbitrary sample point in time $O(log\ N)$.

Unfortunately, the constant of proportionality in their method again depends exponentially on the dimensionality of the underlying search space. As discussed in the previous section, in a high-dimensional space this exponential dependence is apt to make their method impractical. The authors state that one of their goals is to develop nearest neighbor search algorithms that can be applied effectively in spaces of dimensionality up to approximately $k = 20$.

A.4 Multidimensional Search Trees

A *k-d tree* [Bent 1975] is a generalization of a binary search tree to a space of arbitrary dimensions. Each internal node in a k-d tree partitions a subset of points into left and right sub-trees, based on whether a particular coordinate of each point is larger than or smaller than a discriminator associated with the node. A k-d tree generally requires time $O(kN\ log\ N)$ to construct.

When describing the structure of a k-d tree, it is useful to refer to the root as being on the first level, its children as being on the second level and so on. In the original k-d tree as described in [Bent 1975], the root node partitions the entire set of points with respect to the first coordinate in the space. Nodes at levels two through k partition their points with respect to the second through kth coordinates of the space. Nodes on the $(k + 1)$th level cycle back and again partition with respect to the first coordinate. The use of an optimized form of k-d tree to solve the nearest neighbor search problem is discussed in [Frie 1977].

Friedman et al. show that for a large class of underlying distributions, if T contains a sufficient number of points, then a k-d tree can be used to find the nearest neighbor of an arbitrary point in expected time $O(log\ N)$. Friedmann et al. do not give a precise statement of how many points T must contain for asymptotic behavior to occur but give empirical results for the case where the points in T are normally distributed with a unit dispersion matrix. In a 2-dimensional space, on the order of 100 points are sufficient for asymptotic $O(log\ N)$ search times to be achieved. In a 4-dimensional space, on the order of 10,000 points are required to achieve asymptotic behavior. In an 8-dimensional space, the largest data set they tested consisting of 16,000 points was not sufficient for asymptotic behavior to occur.

In [Spro 1991], an experiment using a modified k-d tree was conducted with $k = 16$ and with set T containing 75,857 points. The points in T were generated using a uniform distribution. Sproull reports that in this case, finding the nearest neighbor of a single sample point required on average the examination of more than 95% of the points in the training set. He notes, however, that the efficiency of a k-d tree can improve substantially for non-uniform or "clumpy" data.

The number of sample points required for a k-d tree to achieve logarithmic behavior presumably grows exponentially with the dimensionality of the underlying space. Consequently, for high-dimensional spaces the number of points required for asymptotic behavior is likely to be too large to be achievable in practical applications.

A.5 Similarity Search Using Fourier Coefficients as Indices

As we have discussed in this report, one area where the need for high-dimensional search arises is in the analysis of time series. A search technique designed specifically for time series analysis is described in [Agra 1993]. Unlike the other methods described in this appendix, the method described in [Agra 1993] is heuristic in the sense that it does not have guaranteed worst case or average case sublinear behavior.

The time series considered in [Agra 1993] each contain between 256 and 2,048 elements and thus correspond to points in a space of between 256 and 2,048 dimensions. Given a set *T* of time series, the goal of this method is to find all the elements of *T* which lie within a distance ε of a sample time series *V*.

Agrawal et al. conclude that the dimensionalities required for time series similarity search are much too high for traditional multidimensional search techniques to be directly applied. Instead, Agrawal et al. propose the use of the Discrete Fourier Transform as a means of mapping time series into a lower dimensional space. Given a time series *V*, their method begins by using the Discrete Fourier Transform to transform *V* into a series V^* in the frequency domain. Once V^* has been generated, the first *n* components of V^* are used to map V^* to an *n*-dimensional space. In the examples they discuss, the resulting space typically has a dimensionality of between 2 and 4. Because the dimensionality of the resulting space is so low, a number of efficient multidimensional search techniques can be applied.

Agrawal et al. demonstrate that their method is complete in the sense that it will never fail to find all the points within a given search radius. They do not, however, guarantee that their approach will always be more efficient than exhaustive search. While the method described in [Agra 1993] cannot be guaranteed to be efficient, the authors provide empirical results showing that their method performs significantly better than exhaustive search in particular application domains.

A dimensionality reduction method such as the one described in [Agra 93] can potentially complement the use of the Terasys for nearest neighbor search. Given a high-dimensional data set, a dimensionality reduction technique could be used to reduce the original data to the lowest dimensional space that still permits effective search. If the dimension of the resulting space is sufficiently low, conventional multidimensional search techniques can be used. If the dimension of the resulting space is still too high for conventional techniques, the Terasys could potentially be employed.

LIST OF REFERENCES

[Agra 1993] R. Agrawal, C. Faloutsos, and A. Swami, "Efficient Similarity Search in Sequence Databases," *Proc. FODO Conference*, October 1993.

[Arya 1993] S. Arya and D. M. Mount, "Approximate Nearest Neighbor Queries in Fixed Dimensions," *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, 1993, pp. 271-280.

[Arya 1994] S. Arya, D. M. Mount, N. S. Netanyahu, and R. Silverman, "An Optimal Algorithm for Approximate Nearest Neighbor Searching," *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, 1994, pp. 573-582.

[Bent 1975] J. L. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, Vol. 18, No. 9 (September 1975), pp. 509-517.

[Bent 1980] J. L. Bentley, B. W. Weide, and A. C. Yao, "Optimal Expected-Time Algorithms for Closest Point Problems," *ACM Transactions on Mathematical Software*, Vol. 6, No. 4 (December 1980), pp. 563-580.

[Cove 1967] T. M. Cover and P. E. Hart, "Nearest Neighbor Pattern Classification," *IEEE Transactions on Information Theory*, Vol. IT-13, No. 1 (January 1967), pp. 21-27. Also in [Dasa 1991].

[Cree 1992] R. H. Creecy, B. M. Masand, S. J. Smith, and D. L. Waltz, "Trading MIPS and Memory for Knowledge Engineering," *Communications of the ACM*, Vol. 35, No. 8 (August 1992), pp. 48-63.

[Dasa 1991] B. V. Dasarathy, *Nearest Neighbor (NN) Norms: NN Pattern Classification Techniques,* IEEE Computer Science Press, Los Alamitos, CA, 1991.

[Fix 1951] E. Fix and J. L. Hodges, *Discriminatory Analysis: Nonparametric Discrimination: Consistency Properties*, Project 21-49-004, Report Number 4, USAF School of Aviation Medicine, Randolph Field, TX, 1951, pp. 261-279. Also in [Dasa 1991].

[Fix 1952] E. Fix and J. L. Hodges, *Discriminatory Analysis: Nonparametric Discrimination: Small Sample Performance*, Project 21-49-004, Report Number 11, USAF School of Aviation Medicine, Randolph Field, TX, 1951, pp. 280-322. Also in [Dasa 1991].

[Frie 1975] J. H. Friedman, F. Baskett, and L. J. Shustek, "An Algorithm for Finding Nearest Neighbors," *IEEE Transactions on Computers*, Vol. C-24, No. 10 (October 1975), pp. 1,000-1,006.

[Frie 1977] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time," *ACM Transactions on Mathematical Software*, Vol. 3, No. 3 (September 1977), pp. 209-226.

[Fuku 1990] K. Fukunaga, *Introduction to Statistical Pattern Recognition*, Second Edition, Academic Press, Boston, MA, 1990.

[Gate 1972] G. W. Gates, "The Reduced Nearest Neighbor Rule," *IEEE Transactions on Information Theory*, Vol. IT-18, No. 3 (May 1972), pp. 431-433.

[Gokh 1995] M. Gokhale, B. Holmes, and K. Iobst, "Processing in Memory: The Terasys Massively Parallel PIM Array," *IEEE Computer*, Vol. 28, No. 4 (April 1995), pp. 23-31.

[Hart 1968] P. E. Hart, "The Condensed Nearest Neighbor Rule," *IEEE Transactions on Information Theory*, Vol. IT-14, No. 3 (May 1968), pp. 515-516.

[Marc 1994] J. J. Marciniak, ed., "Encyclopedia of Software Engineering," John Wiley & Sons, New York, NY, 1994.

[Pres 1992] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C*, Cambridge University Press, Cambridge, UK, 1992.

[Shas 1990] D. Shasha and T. Wang, "New Techniques for Best-Match Retrieval," *ACM Transactions on Information Systems*, Vol. 8, No. 2 (April 1990), pp. 140-158.

[Spro 1991] R. F. Sproull, "Refinements to Nearest-Neighbor Searching in *k*-Dimensional Trees," *Algorithmica*, Vol. 6 (1991), pp. 579-589.

[Stan 1986] C. Stanfill and B. Kahle, "Parallel Free-Text Search on the Connection Machine System," *Communications of the ACM*, Vol. 29, No. 12 (December 1986), pp. 1,229-1,239.

LIST OF ACRONYMS

dbC	Data-Parallel, Bit C
IDA	Institute for Defense Analyses
K	1,024
MHz	Megahertz
PIM	Processor in Memory
RNN	Reduced Nearest Neighbor
SIMD	Single Instruction, Multiple Data
VCU	Vecter Comparison Unit

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE August 1996	3. REPORT TYPE AND DATES COVERED Final

4. TITLE AND SUBTITLE Nearest Neighbor Search Applications for the Terasys Massively Parallel Workstation	5. FUNDING NUMBERS IDA Central Research Program (CRP) 9001-506
6. AUTHOR(S) Eric W. Johnson	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard St. Alexandria, VA 22311-1772	8. PERFORMING ORGANIZATION REPORT NUMBER IDA Paper P-3169
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Institute for Defense Analyses (IDA) 1801 N. Beauregard St. Alexandria, VA 22311-1772	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
11. SUPPLEMENTARY NOTES	
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release, unlimited distribution.	12b. DISTRIBUTION CODE 2A

13. ABSTRACT (Maximum 200 words)

The Terasys workstation is a massively parallel computer developed at IDA's Center for Computing Sciences. This report, based on a project conducted in IDA's Computer and Software Engineering Division, presents an overview of the Terasys workstation and discusses how the Terasys could be applied to the task of nearest neighbor search. The report discusses a number of areas where Terasys-based nearest neighbor search could potentially be applied including nearest neighbor classification, pattern recognition, and estimating intrinsic dimensionality. One area where the use of Terasys-based nearest neighbor search seems particularly promising is in the development of new neighbor classifiers. The report includes experimental results showing that a Terasys with 32,768 processors can perform a particular nearest neighbor search problem up to 69 times faster than a Sun Microsystems 61 MHz Sparcstation-20. To illustrate how the Terasys could be applied to the kinds of government work conducted by IDA divisions outside of the Center for Computing Sciences, the report examines three possible applications of Terasys-based nearest neighbor search to an ongoing radar evaluation project in IDA's Science and Technology Division.

14. SUBJECT TERMS Single Instruction, Multiple Data (SIMD) Computers; Massively Parallel Computers; Nearest Neighbor Search.			15. NUMBER OF PAGES 66
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASSIFICATION OF THIS PAGE Unclassified	19. SECURITY CLASSIFICATION OF ABSTRACT Unclassified	20. LIMITATION OF ABSTRACT SAR

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. Z39-18
298-102